IBM

developerWorks®

Intro to graph databases, Part 1: Graph databases and the CRUD operations

Lauren Schaefer

February 22, 2017

If you’re not familiar with graph databases, using one might sound a bit daunting. Anyone who has taken a course on graph theory can probably attest that the subject is a bit complex. But don’t let your graph theory experience (or lack thereof) keep you away from graph databases. In fact, when you use a fully-managed graph database-as-a-service, you get all of the benefits of graph databases without the complexity. This is part one of the two-part tutorial series Intro to Graph Databases.

View more content in this series

To view this video, **The CRUD operations** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

If you're not familiar with graph databases, using one might sound a bit daunting. Anyone who has taken a course on graph theory can probably attest that the subject is a bit complex. But don't let your graph theory experience (or lack thereof) keep you away from graph databases. In fact, when you use a fully managed graph database-as-a-service, you get all of the benefits of graph databases without the complexity.

So what are the benefits I'm alluding to? Graph databases allow you to model your data and the relationships between your data just as they exist in real life; you don't have to abstract the data into tables and use a combination of keys and joins to represent relationships. Plus, graph databases allow you to easily traverse your graph (follow the connections between your data) without doing expensive joins to bring your data together.

Apache TinkerPop is a popular open source graph computing framework. Users of TinkerPop utilize the Gremlin graph traversal language to interact with their graphs. Creating a graph using TinkerPop requires a lot of setup, so we'll use IBM Graph (you can think of it as TinkerPop-as-a-service) to quickly and easily create our graph. IBM Graph is an easy-to-use, fully managed graph database-as-a-service that enables enterprise applications and is built on TinkerPop.

In this tutorial, you'll discover the answers to common questions about graph databases, such as what they are and why you should care about them. Then you'll learn how to use the Gremlin graph traversal language to perform the CRUD (Create, Read, Update, and Delete) operations by exploring the code for the Lauren's Lovely Landscapes app. You'll then implement a new feature that allows end users to view their orders. Stay tuned for Part 2, where you'll get to explore one of the most popular use cases of graph databases: recommendation engines.

The basics of graph databases

If you're new to graph databases, you probably have a few questions about them:

- What are graph databases?
- How do I model my data in a graph database?
- What are some common use cases for graph databases?

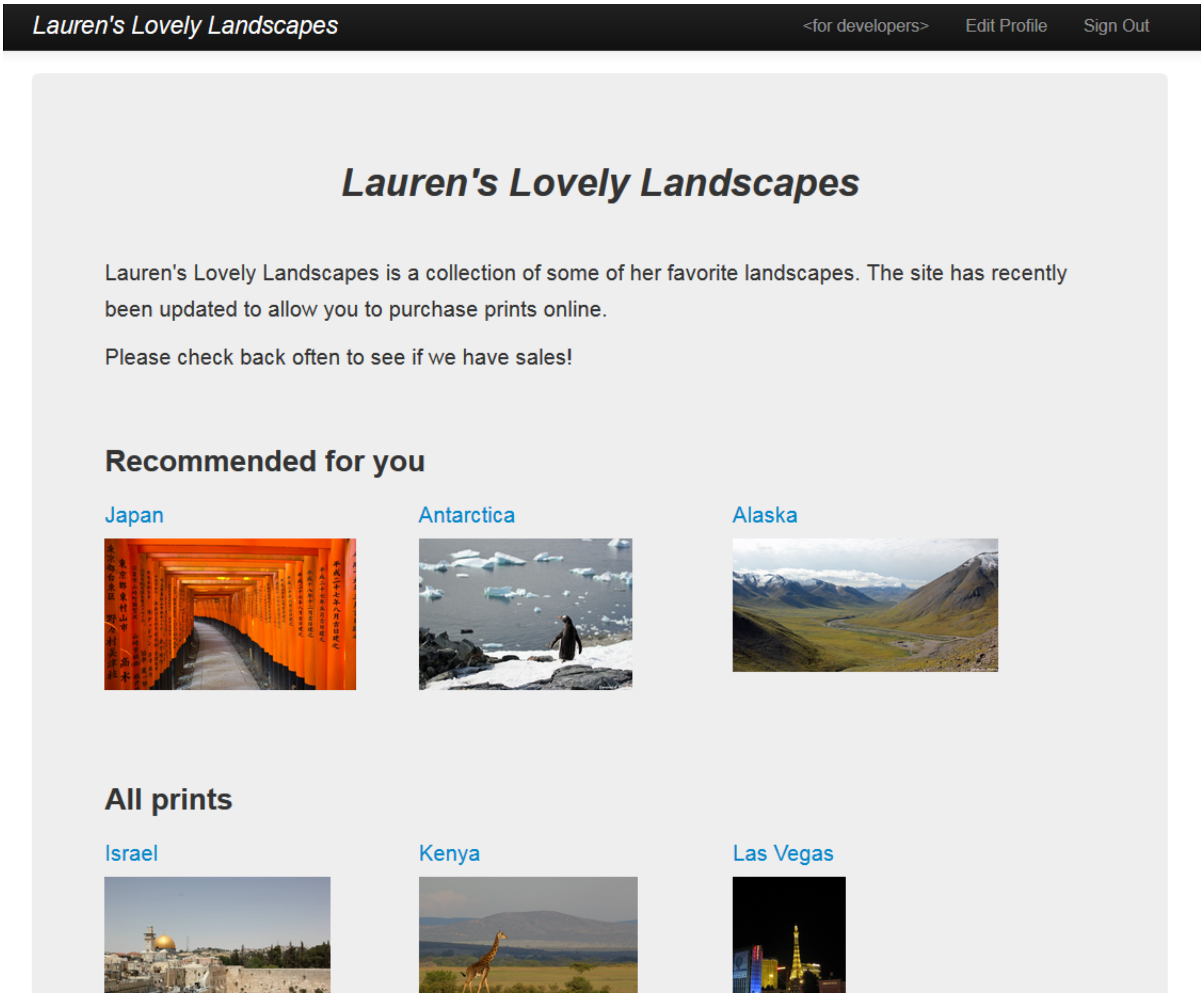
The video "Should I care about graph databases" answers those questions and more. So, sit back, relax, and press play.

To view this video, **Should I care about graph databases?** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

About the app

In this tutorial, you work with a sample online store called Lauren's Lovely Landscapes, which allows you to browse and purchase prints. The **<for developers>** page has information about how the app was built and has links to so you can insert and delete sample data.

Note: This tutorial uses version 2 of Lauren's Lovely Landscapes. If you have an older copy of it from another tutorial, please get the new version.



Lauren's Lovely Landscapes

Lauren's Lovely Landscapes is a collection of some of her favorite landscapes. The site has recently been updated to allow you to purchase prints online.

Please check back often to see if we have sales!

Recommended for you

Japan

Antarctica

Alaska

All prints

Israel

Kenya

Las Vegas

Run the app
Browse the code

Deploy to IBM Bluemix

What you need to get started

Before you begin, you need to register at Bluemix. You'll also need the latest version of one of the following browsers:

- Chrome
- Firefox
- Internet Explorer
- Safari

Deploy the app

To complete the tutorial, you'll want your own copy of the code to edit and deploy. The easiest way to do this without installing anything on your machine is to deploy the app to Bluemix. For more information, see the video Deploy an IBM Graph app to Bluemix.

To deploy the app:

1. Click the following button:
   Deploy to Bluemix
2. If you are not already authenticated at Bluemix, you might be prompted to do so.
3. Click **Deploy** when the button enables. Bluemix will load the code into a Git repo, create an IBM Graph instance that your app will use, and deploy the app to Bluemix.
4. Click the **Delivery Pipeline** tile so you can watch the app deploy.
5. After the Deploy Stage has passed, return to the Toolchain.
6. Click **View app**.
7. The home page shows that no prints are currently available for sale. Click **<for developers>** and then **Insert the sample data**. It might take a minute or two for the data to insert.
8. When the app indicates the sample data has been created, click **Lauren's Lovely Landscapes** in the top navigation bar.
9. Notice the prints listed on the home page. Your app is successfully deployed!

Take a few minutes to explore the app. Register as a new user and order a print. Navigate to the **<for developers>** page to see the schema diagram as well as the data stored in the graph.

Explore the code and existing CRUD operations

In this section, you'll explore the code behind Lauren's Lovely Landscapes to see how the app performs the CRUD (Create, Read, Update, and Delete) operations. As you'll see below, the app POSTs queries to the Gremlin API using the Gremlin graph traversal language to tell Graph what to create, read, update, and delete.

Bonus: To see how the app uses the Graph APIs to create the schema and indexes, watch Create and Traverse an IBM Graph Database.

Open the code

When you deployed the app to Bluemix, a copy of the Lauren's Lovely Landscapes code was automatically put in a project for you. In this section, you'll open the code in the Bluemix web IDE so you can browse the code in the sections below:

1. Navigate to Bluemix.net.
2. Log in (if you are not already authenticated).
3. On the dashboard for your apps, locate the row with the Lauren's Lovely Landscapes app and click it.
   Hint: Be sure to click the name of your app and not the route.
4. The app page opens in Bluemix.
5. Scroll down until you see the **Continuous delivery** tile.

6. Click the **View toolchain** button.
7. Click the **Eclipse Orion Web IDE** tile. The web IDE opens with your project's code.

Session API

To make requests to the Graph API, you need to authenticate. One option is to send your credentials with every request. A better, faster option is to use the session API to generate a unique session token to use in later requests as your form of authentication.

Lauren's Lovely Landscapes uses the latter option of generating a unique session token. Let's explore:

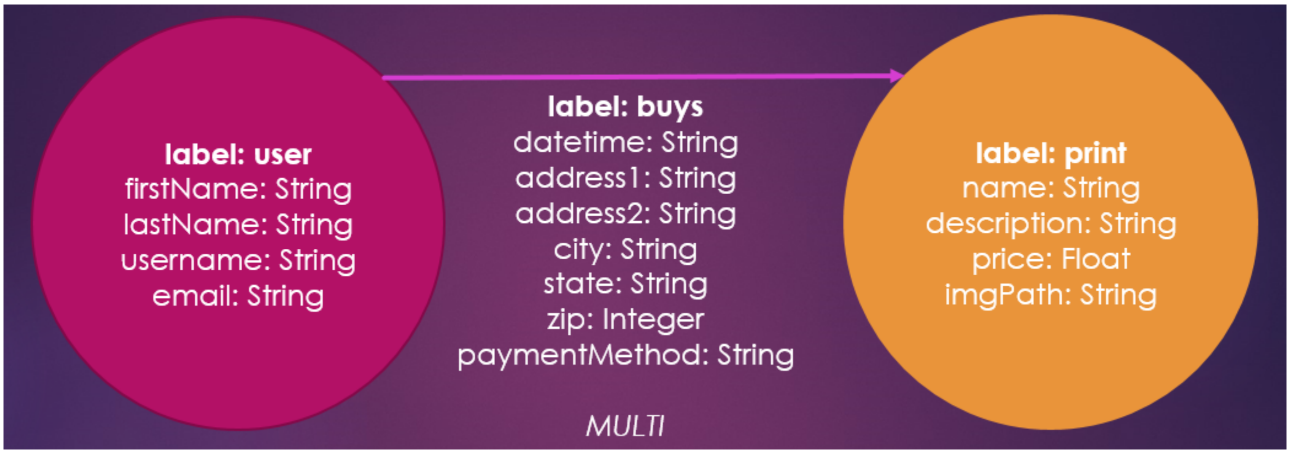
1. In the file navigation pane of the web IDE, click **constants.py** to open it.
2. Note that the code between lines 23 and 31 accesses the credentials stored in VCAP_SERVICES to acquire the apiURL, username, and password from Bluemix. These credentials are used later to authenticate and get your session token.
   Hint: if you choose to deploy your app locally instead of on Bluemix, follow the instructions in the code around line 11 to manually set the API_URL, USERNAME, and PASSWORD.
3. In the file navigation pane of the web IDE, click **graph.py** to open it.
4. In **graph.py**, locate the **updateToken()** function around line 400. This function makes a request to the Session API, sending over the username and password that were acquired in **constants.py**. The response from the session API includes the unique token needed for authentication in later requests to the Graph API. The function adds the token to the headers. Note that this is the only function that uses `constants.USERNAME` and `constants.PASSWORD`. All other calls use the unique session token.
5. While still in **graphy.py**, locate the **post()** function around line 11 and the **get()** function around line 19. These functions handle GET and POST requests to the Graph API. Note that if these functions receive a 401 error response (because the token expires) or a 403 error response (because the credentials have been invalidated), the functions automatically makes a request to update the token.
6. Remaining in **graphy.py**, locate the **getAllPrints()** function around line 27. This function gets all of the prints stored in the graph. Note that this function (as do all of the other functions in this file that make GET requests to the Graph API) calls the **post()** function we just explored. The **getAllPrints()** function doesn't bother with authentication or headers; it simply focuses on the API call and the response.

For more details on the Session API, see the IBM Graph Documentation.

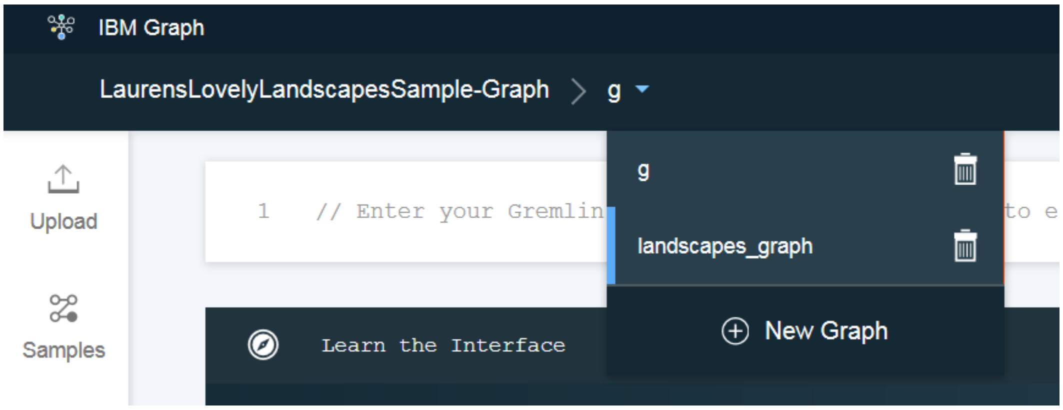
Create

You now understand how the API calls authenticate using a unique session token. Time for the fun stuff: The CRUD operations, starting with create. The following instructions guide you through how the app creates a new vertex in the graph when a user registers. For more information, see the video Create data elements in IBM Graph.

1. Observe the schema diagram below. To handle user registration, create a new user **vertex** with the following properties: **firstName**, **lastName**, **username**, and **email**.



2. Next, open the Graph Query Editor for your graph instance in a new browser tab or window:
    a. Navigate to Bluemix.net.
    b. On the dashboard, scroll down to the All Services section and click **LaurensLovelyLandscapesSample-Graph**.
    c. On the Manage tab (open by default), click **Open**. The Graph Query Editor opens for your graph instance.
3. By default, the **g** graph is selected. Switch to the **landscapes_graph** where your data is stored by clicking the down arrow beside **g** in the top navigation menu and clicking **landscapes_graph**.



4. To create a new user, write a Gremlin query that includes sample data for each of the properties shown in the schema diagram: **label**, **firstName**, **lastName**, **username**, and **email**. Including a property **type** with an associated value **user** makes it possible to search for vertexes with type **user**. In the Query Execution Box, type the following Gremlin query:

```
def gt = graph.traversal();
gt.addV(label, 'user', 'firstName', 'Lauren', 'lastName', 'Schaefer', 'username',
'lauren', 'email', 'lauren@example.com', 'type', 'user');
```

5. Click the Submit Query button (⊛).
6. The results of the query open in a new box. Explore the JSON results. Note that a new vertex has been created using the properties we indicated in the query.
7. Now that you have a working query that creates a new **user** vertex, explore the code. In the file navigation pane of the web IDE you left open in another browser tab or window, click **graph.py** to open it.

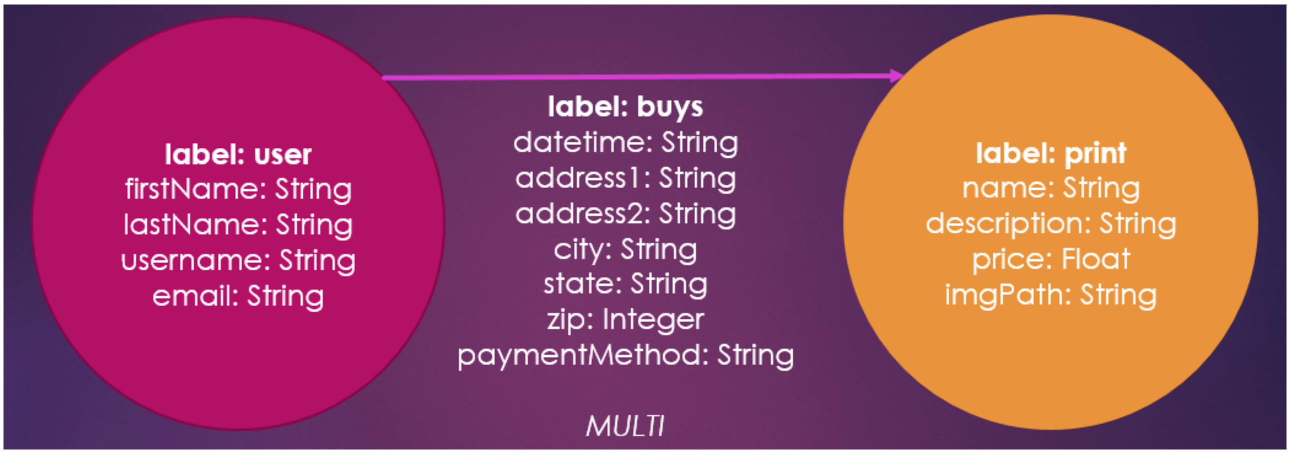
8. In **graph.py**, locate the **createUser()** function around line 258.
9. The function begins by calling the **doesUserExist()** function to check if a user with the given username already exists. The function returns an error that is displayed to the user if the username they requested is already taken.
10. If the username is available, the function continues and creates a new dictionary that contains a Gremlin query. The query is based on the one we wrote above in step 4; the only difference is that instead of using sample data for the properties **firstName**, **lastName**, **username**, and **email**, the code uses dynamic input based on the arguments passed in to the function.
11. After the dictionary containing the query is created, the function is ready to call the Graph API. Around line 278, the function makes a new POST request to `/gremlin` and sends the dictionary containing the Gremlin query as part of the request.
12. Around line 280, the function checks to see if the request was successful (200 response code) and the user vertex was created. If the request was not successful, the function raises an error.

For more information on the Gremlin API, see the IBM Graph Documentation.

Read

In this section, you'll explore the read operation. The following instructions guide you through how the app reads a user vertex when displaying a user's profile information. For more information, see the video Read data elements in IBM Graph.

1. Let's begin by observing the schema diagram. To display a user's profile information, you need to read a user vertex with its properties.



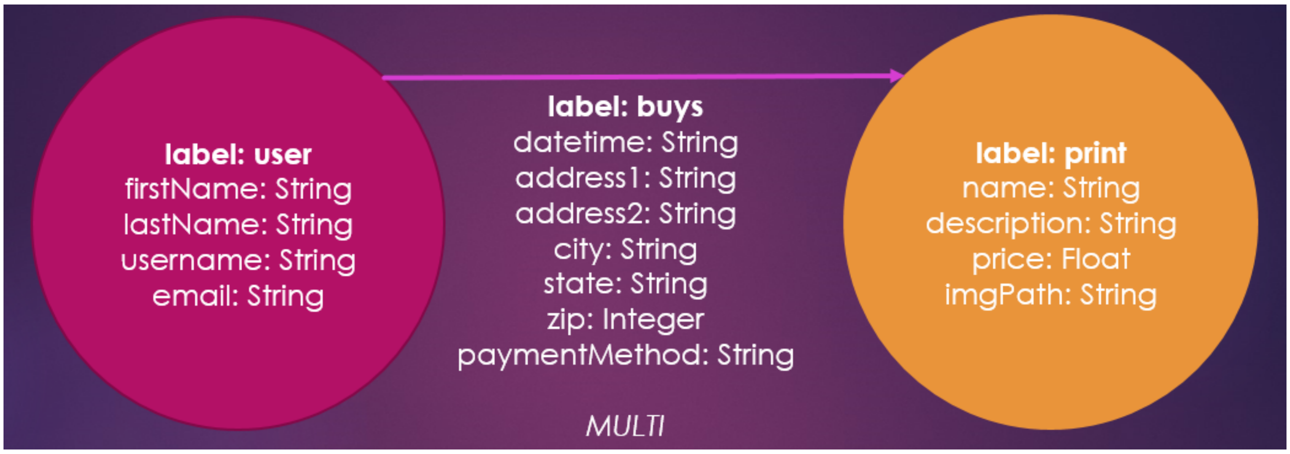
2. Open your browser tab or window that has the Graph Query Editor (instructions for how to do this are in Create > Step 2.) Ensure the landscapes_graph is selected (see Create > Step 3 for instructions on how to do this.)
3. Write a Gremlin query to read a user vertex with the label **user** and the username "jason." To do this, in the Query Execution Box, type the following Gremlin query:
```
def gt = graph.traversal();
gt.V().hasLabel("user").has("username", "jason");
```
4. Click the Submit Query button (⊕).

5. The results of the query open in a new box below. Explore the JSON results. Note that the results are returned as a set of length one. This is because only one vertex has the username "jason."
6. Now that you have a working query that reads a user vertex, explore the code. In the file navigation pane of the web IDE you left open in another browser tab or window, open **graph.py**.
7. In **graph.py**, locate the **getUser()** function around line 204.
8. The function begins by creating a new dictionary that contains a Gremlin query. The query is based on the one you wrote in step 3 above; the only difference is that instead of querying for username "jason," the code uses dynamic input based on the username passed in to the function.
9. After the dictionary containing the query is created, the function is ready to call the Graph API. Around line 208, the function makes a new POST request to `/gremlin` and sends the dictionary containing the Gremlin query as part of the request.
10. Around line 209, the function checks to see if the request was successful (200 response code). Then the function starts processing the results. The JSON results shown in the query editor are found by accessing `json.loads(response.content)['result']['data']`. The results are a set of vertexes, so the function checks that the length of the results is greater than 0. Because only one vertex should be returned when a particular username is queried, the function sets `user` to `results[0]`. If everything was successful, the user vertex with its properties is returned. Otherwise, the function raises an error.

Update

Next, you'll explore the update operation. The following instructions walk you through how the app updates a user vertex when a user makes changes to his profile information. For more information, watch the Update data elements in IBM Graph video.

1. The schema diagram is shown below. To edit a user's profile information, update the properties stored in a user vertex.



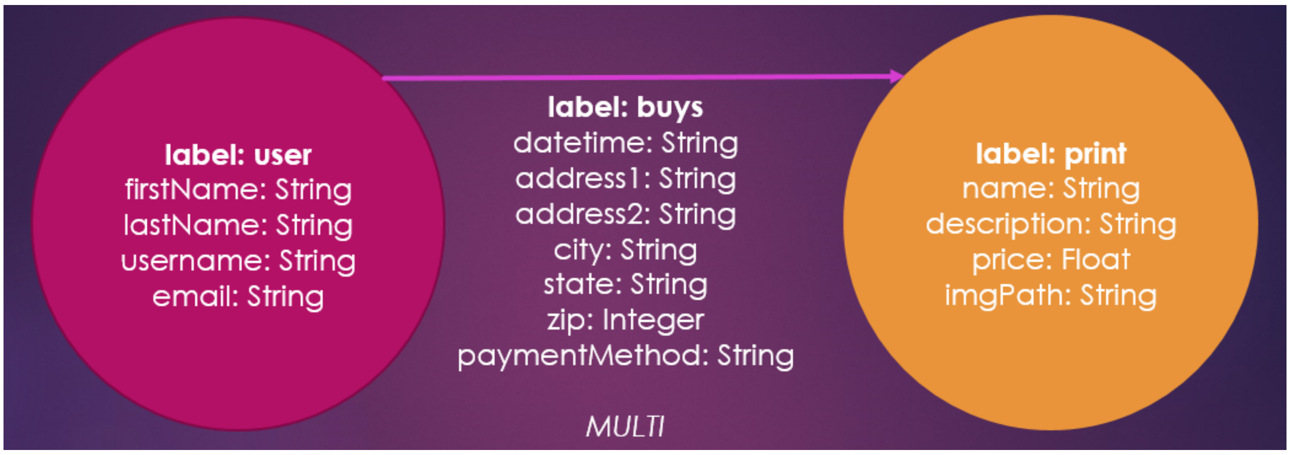
2. Open your browser tab or window that has the Graph Query Editor (instructions for how to do so are in Create > Step 2.) Ensure the landscapes_graph is selected (instructions for how to do so are in Create > Step 3.)

3. Let's write a Gremlin query to update a user vertex. We'll query for the vertex with label **user** and the username "jason" and update that vertex with new property values. In the Query Execution Box, input the following Gremlin query:
   ```
   def gt = graph.traversal();
   gt.V().hasLabel("user").has("username", "jason").property('firstName',
   'Jasonupdate').property('lastName', 'Schaeferupdate').property('email',
   'jasonupdate@example.com');
   ```
4. Click the Submit Query button (➔).
5. The results of the query open in a new box below. Explore the JSON results. Note that the "Jason" vertex now has the property values we used in the query.
6. Now that you have a working query that updates a user vertex, explore the code. In the file navigation pane of the web IDE you left open in another browser tab or window, open **graph.py**.
7. In **graph.py**, locate the **updateUser()** function around line 219.
8. The function begins by creating a new dictionary that contains a Gremlin query. The query is based on the one you wrote above in step 3 with a few differences: instead of querying for username "jason" and updating the properties with sample data, the code uses dynamic input based on the arguments passed in to the function.
9. After the dictionary containing the query is created, the function is ready to call the Graph API. Around line 228, the function makes a new POST request to `/gremlin` and sends the dictionary containing the Gremlin query as part of the request.
10. Around line 229, the function checks to see if the request was successful (200 response code) and the user vertex was updated. If the request was not successful, the function raises an error.

Delete

Finally, let's explore the delete operation. The following instructions guide you through how the app deletes all of the edges and vertexes in the graph. For more information, see the video Delete data elements in IBM Graph.

1. Let's begin by observing the schema diagram. To delete all of the vertexes and edges in the graph, you'll need to delete all of the **buys** edges as well as the **user** and **print** vertexes.



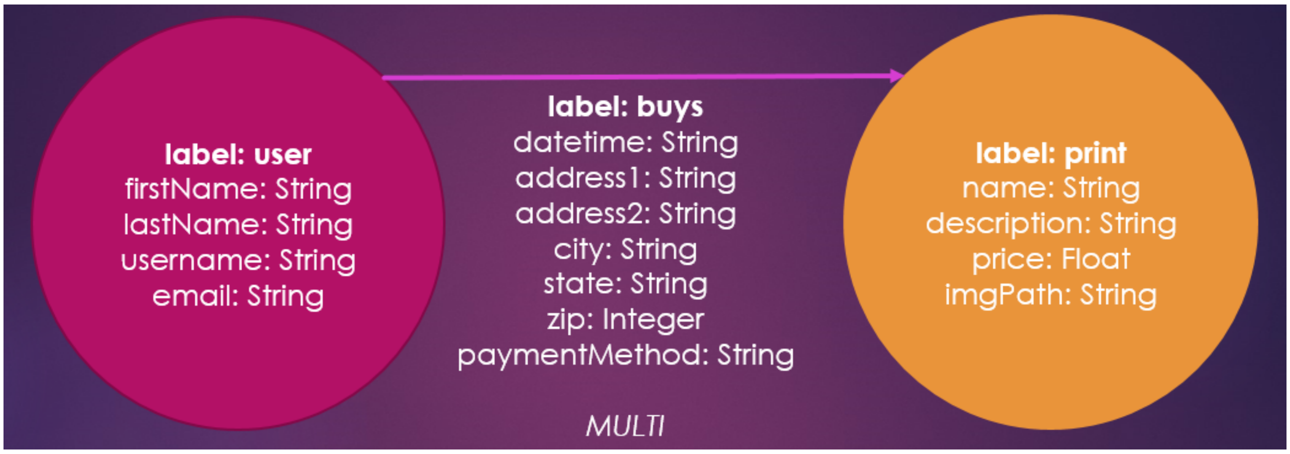
2. Open your browser tab or window that has the Graph Query Editor (instructions for how to do so are in Create > Step 2.) Ensure the landscapes_graph is selected (instructions for how to do so are in Create > Step 3.)
3. Write a Gremlin query to delete all of the edges and vertexes. Query all of the edges of type **buys** and drop them. Then query all of the vertexes of type **print** or **user** and drop them. In the Query Execution Box, input the following Gremlin query:
   ```
   def g = graph.traversal();
   g.E().has('type', 'buys').drop();
   g.V().has('type', within('print','user')).drop();
   ```
4. Click the Submit Query button (➔).
5. The results of the query open in a new box below. Note that an empty list is displayed as the query has dropped all of our vertexes and edges.
   Hint: If you want to interact with your copy of the sample app, return to the **<for developers>** page and insert the sample data.
6. Now that you have a working query that deletes all of the edges and vertexes, let's explore the code. In the file navigation pane of the web IDE you left open in another browser tab or window, click **graph.py** to open it.
7. In **graph.py**, locate the **dropGraph()** function around line 441.
8. The function begins by creating a new dictionary that contains a Gremlin query. The query is based on the one we wrote above in step 3.
9. After the dictionary containing the query is created, the function is ready to call the Graph API. Around line 447, the function makes a new POST request to `/gremlin` and sends the dictionary containing the Gremlin query as part of the request.
10. Around line 448, the function checks to see if the request was successful (200 response code) and the edges and vertexes were deleted. If the request was not successful, the function raises an error.

Implement a read operation

Now that you know the basics of how to perform the CRUD operations using Gremlin, it's time to code. In this section, you'll update the user profile page to display the user's orders:

1. Start by observing the schema diagram. To view the orders for a user, you'll need to start with the **user** vertex for the authenticated user and then traverse the graph to read property values stored in the **buys** edge (**datetime**, **address1**, **address2**, **city**, **state**,

zip, and **payment method**) and the associated **print** vertex (**name** and **imgPath**).



2. Open your browser tab or window that has the Graph Query Editor (instructions for how to do so are in Create > Step 2.) Ensure the landscapes_graph is selected (instructions for how to do so are in Create > Step 3.)
3. You'll need to write a Gremlin query to read all of the information about a given user's orders. To do this, start by querying for the vertex with label **user** and the username "jason." Name this vertex "buyer." Then traverse out along the buys edge to find all of the print vertexes that the user has bought. Next, search for all of the **in edges** that connect to the buyer vertex (if you fail to check that the vertex is the buyer vertex, the query will return all connected edges regardless of who bought the print). Finally, use the `path` step so you can view the history of the traversal that includes the **user** vertex, the **buys** edge, and the **print** vertex (if you don't use the `path` step, the results will only contain the final **user** vertex). In the Query Execution Box, type the following Gremlin query:
```
def gt = graph.traversal();
gt.V().hasLabel("user").has("username", "jason").as('buyer')
.out("buys")
.inE("buys").outV().as('buyer2').where('buyer',eq('buyer2'))
.path();
```
4. Click the Submit Query button (➔).
5. The results of the query open in a new box. From the diagram on the right, you can see the user has bought three prints. The JSON results are displayed on the left. Each JSON object contains the information for one order. The `objects` property contains the set of information we care about: the **user** vertex, the **print** vertex, the **buys** edge, and a duplicate **user** vertex the traversal found because it started and ended by searching for a **user** vertex.
Hint: If no results are returned, be sure you inserted the sample data on the **<for developers>** page.
6. Now that you have confirmed the query works, let's code. In the file navigation pane of the web IDE you left open in another browser tab or window, open **graph.py**.
7. Beneath the `getUser()` function around line 217, paste the following code:

```
def getUserOrders(username):
    gremlin = {
        "gremlin": "def gt = graph.traversal();" +
            "gt.V().hasLabel(\"user\").has(\"username\", \"" + username + "\").as(\"buyer\")" +
            ".out(\"buys\")" +
            ".inE(\"buys\")" +
            ".outV().as(\"buyer2\").where(\"buyer\",eq(\"buyer2\"))" +
            ".path()"
        }
    response = post(constants.API_URL + '/' + constants.GRAPH_ID + '/gremlin', json.dumps(gremlin))
    if (response.status_code == 200):
        results = json.loads(response.content)['result']['data']
        orders = []
        if len(results) > 0:
            print 'Found orders for username %s: %s.' % (username, results)

            for result in results:
                order = {}
                for object in result['objects']:
                    if object['label']=='user':
                        continue
                    if object['label']=='buys':
                        order['date'] = object['properties']['date']
                        order['firstName'] = object['properties']['firstName']
                        order['lastName'] = object['properties']['lastName']
                        order['address1'] = object['properties']['address1']
                        order['address2'] = object['properties']['address2']
                        order['city'] = object['properties']['city']
                        order['state'] = object['properties']['state']
                        order['zip'] = object['properties']['zip']
                        order['paymentMethod'] = object['properties']['paymentMethod']
                        continue
                    if object['label']=='print':
                        order['printName'] = object['properties']['name'][0]['value']
                        order['imgPath'] = object['properties']['imgPath'][0]['value']
                        continue

                orders.append(order)

        return orders

    raise ValueError('Unable to find orders for user with username %s' % username)
```

Note: Spacing is very important when programming in Python. Be sure you use spaces to appropriately indent the code.

Let's examine what this function does. First, the code creates a dictionary that holds the Gremlin query. This query is similar to the one used in step 3 above, except instead of querying for the username "jason," the code uses dynamic input based on the argument passed in to the function. Second, the code includes the query in a POST request to the Gremlin API. Third, the code processes the results. If the response is 200, the query was successful. The JSON results just as we saw in the query editor can be found by accessing `json.loads(response.content)['result']['data']`, so the code stores them in results. The code creates a new list, named **orders,** where all of the information that will be displayed to the end user about their orders is stored. Then the code starts looping through each **object** in each **result**. The code checks the label of each object (**user**, **buys**, or **print**) and then stores the appropriate properties in **orders**. Finally, if everything goes well, the code returns **orders**. If not, the code raises a **ValueError**.

8. Now that the back-end code is done, you need to get this order information to the front end. In the left navigation pane in the web IDE, click **wsgi.py** to open it.
9. Locate the **getProfile()** function around line 189. This function is called whenever a user accesses the profile page. The last line of the function returns the template for the profile

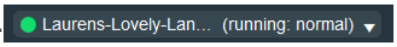
page. Update the arguments that are sent to `bottle.template()` so that the orders are included:
`return bottle.template('profile', username = username, userInfo = graph.getUser(username), orders = graph.getUserOrders(username))`
Hint: Be sure the spacing before `return` remains the same.

10. Now that the profile template has access to the orders, you need to update it to display the orders. In the left navigation pane in the web IDE, expand the **views** directory and click **profile.tpl** to open it.
11. After the closing tag of the form but before the line to include the footer around line 54, paste the following code:

```
<h2><em>Your Orders</em></h2>
% if len(orders) <= 0:
  You have not placed any orders.
% else:
  % for order in orders:
    <hr>
 <div class="container">
  <div class="row">
   <div class="span6">
    <h3>Order date:</h3>
    {{order['date']}}

    <h3>Shipping address:</h3>
    {{order['firstName']}} {{order['lastName']}}<br>
    {{order['address1']}}<br>
    % if len(order['address2']) > 0:
     {{order['address2']}}<br>
    % end
    {{order['city']}}, {{order['state']}} {{order['zip']}}

    <h3>Payment method:</h3>
    {{order['paymentMethod']}}
   </div>

   <div class="span4">
    <h3>Print:</h3>
    {{order['printName']}}
    <img src="/static/images/{{order['imgPath']}}">
   </div>

  </div>
 </div>

  % end
% end
```

Let's examine what this code does. The code begins by creating a second-level heading titled Your Orders. The code then checks to see if the user has any orders. If the user does not have any orders, the code displays an appropriate message to the user. If the user has orders, the code loops through them. For each order, it creates a new container and row that is split into two divs. The left div displays information about the order and the right div displays information about the print.

12. You have finished implementing our code! Before you celebrate, let's test it by deploying the app. Open **manifest.yml**. On line 11, replace **LaurensLovelyLandscapesSample-${random-word}** with the host for your app. For example, if the url for your app

is https://laurenslovelylandscapessample-cuneal-freesheet.mybluemix.net/, type `laurenslovelylandscapessample-cuneal-freesheet` as your host.
13. In the toolbar at the top of the page, click **Create new launch configuration** and then click the button to create a new launch configuration (⊞).
14. Click **Save**.
15. In the toolbar at the top of the web IDE, click the Deploy the App from the Workspace button (▶). The deploy can take a minute or two. When the app has finished deploying, a green status dot appears beside your app's name in the toolbar. 
● Laurens-Lovely-Lan... (running: normal) ▾
16. When the app finishes deploying, click the Open the Deployed App button (↗) in the toolbar at the top of the web IDE. Your deployed version of Lauren's Lovely Landscapes opens.
17. In your deployed app, sign in with username "jason" (if not already authenticated).
18. Click **Edit Profile**.
19. Observe the new **Your Orders** section you just implemented. Now it's time to celebrate!

What's next?

Before we get ahead of ourselves, let's reflect on all that you learned and did in this tutorial. You began by learning about graph databases and their common use cases. Then you deployed your own copy of the Lauren's Lovely Landscapes app to Bluemix and learned how you can use Graph's Session API to generate a unique session token for authentication. Next, you explored how to perform each of the CRUD operations using Gremlin and the Graph APIs. Finally, you implemented a new feature to allow users to view their orders by leveraging the read operation. You even got to see your new feature running live on Bluemix.

Now that you know the basics, you're ready for the exciting stuff: Recommendation engines. Continue to Part 2, where you'll learn about the existing recommendation engine and how to build your own.

Related topics

- Intro to IBM Bluemix DevOps Services
- IBM Graph – From Zero to Insights
- How can I try out graph databases?
- IBM Graph Documentation
- IBM Graph Learning Center
- The New Builders Ep. 12: Of Graphs and Gremlins – Graph Database 101
- The New Builders Ep. 28: Tinkering with a Graph Database Service